Metodologías de Programación II
Listas en Lisp

Dr. Alejandro Guerra-Hernández

Departamento de Inteligencia Artificial
Facultad de Física e Inteligencia Artificial
aguerra@uv.mx
http://www.uv.mx/aguerra

Maestría en Inteligencia Artificial 2012

LISP = LISt Processor

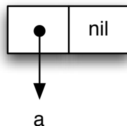
- Las listas fueron originalmente la principal estructura de datos en Lisp.
- El nombre del lenguaje es un acrónimo de "LISt Processing".
- Esta sesión muestra qué es lo que uno puede hacer con las listas y las usa para introducir algunos conceptos generales de Lisp.

Universidad Veracruzana

Definición

- Lo que `cons` hace es combinar dos objetos, en uno formado de dos partes llamado *cons*.
- Un *cons* es un par de apuntadores: el primero es el *car* y el segundo el *cdr*.

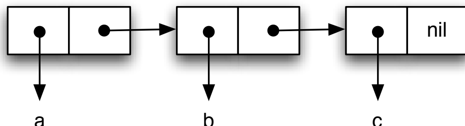
```
1  CL-USER> (cons 'a nil)
2  (A)
3  CL-USER> (car '(a))
4  A
5  CL-USER> (cdr '(a))
6  NIL
```



Listas y Conses

- Cuando construimos una lista con múltiples componentes, el resultado es una cadena de *conses*.

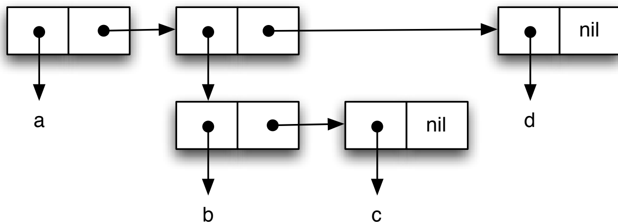
```
1  CL-USER> (list 'a 'b 'c)
2  (A B C)
3  CL-USER> (cdr (list 'a 'b 'c))
4  (B C)
```



Listas y Conses anidados

- Esto también aplica a las listas anidadas:

```
1  CL-USER> (list 'a (list 'b 'c) 'd)
2  (A (B C) D)
```



¿Es lista o átomo?

- La función `consp` regresa *true* si su argumento es un *cons*, así que podemos definir:

```
1   CL-USER> (defun mi-listp (x)
2              (or (null x) (consp x)))
3   MI-LISTP
4   CL-USER> (defun mi-atomp (x)
5              (not (consp x)))
6   MI-ATOMP
7   CL-USER> (mi-listp '(1 2 3))
8   T
9   CL-USER> (mi-atomp '(1 2 3))
10  NIL
11  CL-USER> (mi-listp nil)
12  T
13  CL-USER> (mi-atomp nil)
14  T
```

Cons e Igualdad

- Cada vez que invocamos a *cons*, Lisp reserva memoria para dos apuntadores.
- Si llamamos a *cons* con el mismo argumento dos veces, Lisp regresa dos valores que aparentemente son el mismo, pero en realidad se trata de diferentes objetos:

```
1  CL-USER> (eql (cons 1 nil) (cons 1 nil))
2  NIL
```

Universidad Veracruzana

Iguales si tienen los mismos elementos

- La función `mi-eql` regresa *true* cuando dos listas tienen los mismos elementos:

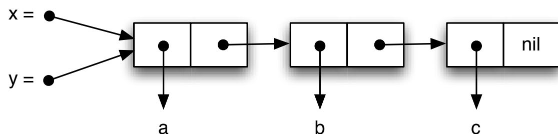
```
1  CL-USER> (defun mi-eql (lst1 lst2)
2      (or (eql lst1 lst2)
3          (and (consp lst1)
4               (consp lst2)
5               (mi-eql (car lst1) (car lst2))
6               (mi-eql (cdr lst1) (cdr lst2)))))
7  MI-EQL
8  CL-USER> (mi-eql (cons 1 nil) (cons 1 nil))
9  T
```

- Esta función es equivalente a `equal`.

Variables, Conses y Apuntadores

- Así como los *conses* tienen apuntadores a sus elementos, las variables tienen apuntadores a sus valores.

```
1  CL-USER> (setf x '(a b c))
2  (A B C)
3  CL-USER> (setf y x)
4  (A B C)
5  CL-USER> (eql x y)
6  T
```



Copiando una lista

- La función `copia-lista` recibe una lista y regresa una copia de ella (en conses diferentes).

```
1   CL-USER> (defun copia-lista (lst)
2              (if (atom lst)
3                  lst
4                (cons (car lst) (copia-lista (cdr lst)))))
5   COPIA-LISTA
6   CL-USER> (setf x '(a b c)
7                  y (copia-lista x))
8   (A B C)
9   CL-USER> (eql x y)
10  NIL
```

Un algoritmo de compresión de datos

- RLE (*run-length encoding*) es un algoritmo de compresión de datos muy sencilla.
- Funciona como los meseros: si los comensales pidieron una tortilla española, otra, otra más y una ensalada verde; el mesero pedirá tres tortillas españolas y una ensalada verde.

El código

- El código de esta estrategia es como sigue:

```
1  (defun rle (lst)
2    (if (consp lst)
3        (compress (car lst) 1 (cdr lst))
4        lst))
5
6  (defun compress (elt n lst)
7    (if (null lst)
8        (list (n-elts elt n))
9        (let ((sig (car lst)))
10         (if (eql sig elt)
11             (compress elt (+ n 1) (cdr lst))
12             (cons (n-elts elt n)
13                   (compress sig 1 (cdr lst)))))))
14
15 (defun n-elts (elt n)
16   (if (> n 1)
17       (list n elt)
18       elt))
```

Llamada a RLE

- Una llamada a *rle*/1 sería como sigue:

```
1  CL-USER> (rle '(1 1 1 0 1 0 0 0 0 1))
2  ((3 1) 0 1 (4 0) 1)
```

- Programen una función inversa a *rle*: dada una lista que es un código *rle*, regrese la cadena original.
- Observen que este método de comprensión no tiene perdida de información. Prueben su solución con la ayuda de un generador de listas de *n* elementos aleatorios.

Universidad Veracruzana

Biblioteca básica de funciones sobre listas

```
1  (proclaim '(inline last1 single append1 conc1 mklist))
2  (proclaim '(optimize speed))
3
4  (defun last1 (lst)
5    (car (last lst)))
6
7  (defun single (lst)
8    (and (consp lst) (not (cdr lst))))
9
10 (defun append1 (lst obj)
11   (append lst (list obj)))
12
13 (defun conc1 (lst obj)
14   (nconc lst (list obj)))
15
16 (defun mklist (obj)
17   (if (listp obj) obj (list obj)))
```

Universidad Veracruzana

last1

- La función `last1` regresa el último elemento de una lista.
- La función predefinida `last` regresa el último *cons* de una lista, no su último elemento.
- `last1` no lleva a cabo ningún chequeo de error:

```
1  CL-USER> (last1 "prueba")
2  value "prueba" is not of the expected type LIST...
```

- Cuando las utilidades son tan pequeñas, forman una capa de abstracción tan delgada, que son transparentes.

Universidad Veracruzana

single

- La función `single` prueba si algo es una lista de un elemento. Los programas Lisp necesitan hacer esta prueba bastantes veces. Al principio, uno está tentado a utilizar la traducción natural del español a Lisp:

```
1   (= (length lst) 1)
```

- pero escrita de esta forma, la función sería muy ineficiente.

append1 y conc1

- Las funciones `append1` y `conc1` agregan un elemento al final de una lista, `conc1` de manera destructiva.
- Estas funciones son pequeñas, pero se usan tantas veces que vale la pena incluirlas en la librería.
- De hecho, `append1` ha sido predefinida en muchos dialectos Lisp (pero no en Lispworks).

mklist

- La función `mklst` nos asegura que su argumento sea una lista.
- Muchas funciones Lisp están escritas para regresar una lista o un elemento. Supongamos que `lookup` es una de estas funciones. Si queremos colectar el resultado de aplicar esta función a todos los miembros de una lista, podemos escribir:

```
1  (mapcar #'(lambda (d) (mklist (lookup d)))
2     data)
```

Otras funciones: longer y filter

```
1   (defun longer (x y)
2     (labels ((compare (x y)
3                (and (consp x)
4                     (or (null y)
5                         (compare (cdr x) (cdr y))))))
6       (if (and (listp x) (listp y))
7           (compare x y)
8           (> (length x) (length y)))))
9
10  (defun filter (fn lst)
11    (let ((acc nil))
12      (dolist (x lst)
13        (let ((val (funcall fn x)))
14          (when val (push val acc))))
15      (nreverse acc)))
```

Otras funciones: group

```
1   (defun group (src n)
2     (if (zerop n) (error "la fuente es de longitud 0")
3         (labels ((rec (src acc)
4                    (let ((rest (nthcdr n src)))
5                      (if (consp rest)
6                          (rec rest (cons (subseq src 0 n) acc))
7                          (nreverse (cons src acc))))))
8           (if src (rec src nil) nil))))
```

Corridas

```
1   CL-USER> (filter #'null '(nil t nil t 5 6))
2   (T T)
3   CL-USER> (filter #'(lambda (x)
4                          (when (> x 0) x))
5                      '(-1 2 -3 4 -5 6))
6   (2 4 6)
7   > (filter #'(lambda (x)
8                    (if (numberp x) (1+ x)))
9               '(a 1 2 b 3 c d 4))
10  (2 3 4 5)
11  CL-USER> (group '(a b c d e f g) 2)
12  ((A B) (C D) (E F) (G))
```

Universidad Veracruzana

Funciones doblemente recursivas

```lisp
1  (defun flatten (x)
2    (labels ((rec (x acc)
3               (cond ((null x) acc)
4                     ((atom x) (cons x acc))
5                     (t (rec (car x) (rec (cdr x) acc))))))
6      (rec x nil)))
7
8  (defun prune (test tree)
9    (labels ((rec (tree acc)
10              (cond ((null tree) (nreverse acc))
11                    ((consp (car tree))
12                     (rec (cdr tree)
13                          (cons (rec (car tree) nil) acc)))
14                    (t (rec (cdr tree)
15                            (if (funcall test (car tree))
16                                acc
17                                (cons (car tree) acc)))))))
18     (rec tree nil)))
```

Universidad Veracruzana

Corridas

- La primera de ellas, `flatten` regresa la lista de átomos que son elementos de su lista argumento:

```
1  CL-USER 1 > (flatten '(a (b c) ((d e) f)))
2  (A B C D E F)
```

- La segunda función, `prune` remueve de una lista todo átomo que satisface el predicado `test`, de forma que:

```
1  CL-USER 2 > (prune #'evenp '(1 2 (3 (4 5) 6) 7 8 (9)))
2  (1 (3 (5)) 7 (9))
```

Segundo orden: mapeos

```
1  (defun map0-n (fn n)
2    (mapa-b fn 0 n))
3
4  (defun map1-n (fn n)
5    (mapa-b fn 1 n))
6
7  (defun mapa-b (fn a b &optional (step 1))
8    (do ((i a (+ i step))
9         (result nil))
10        ((> i b) (nreverse result))
11     (push (funcall fn i) result)))
12
13 (defun map-> (fn start test-fn succ-fn)
14   (do ((i start (funcall succ-fn i))
15        (result nil))
16       ((funcall test-fn i) (nreverse result))
17     (push (funcall fn i) result)))
```

Universidad Veracruzana

Más mapeos

```
1   (defun mapcars (fn &rest lsts)
2     (let ((result nil))
3       (dolist (lst lsts)
4         (dolist (obj lst)
5           (push (funcall fn obj) result)))
6       (nreverse result)))
7
8   (defun rmapcar (fn &rest args)
9     (if (some #'atom args)
10        (apply fn args)
11        (apply #'mapcar
12               #'(lambda (&rest args)
13                   (apply #'rmapcar fn args))
14               args)))
```

Corridas

```
1  CL-USER 3 > (map0-n #'1+ 5)
2  (1 2 3 4 5 6)
3  CL-USER 4 > (map1-n #'1+ 5)
4  (2 3 4 5 6)
5  CL-USER> (mapa-b #'1+ 1 4 0.5)
6  (2 2.5 3.0 3.5 4.0 4.5 5.0)
7  CL-USER 11 > (rmapcar #'princ '(1 2 (3 4 (5) 6) 7 (8 9)))
8  123456789
9  (1 2 (3 4 (5) 6) 7 (8 9))
10 CL-USER 12 > (rmapcar #'+ '(1 (2 (3) 4)) '(10 (20 (30) 40)))
11 (11 (22 (33) 44))
```

mapa-b en términos de map->

```
1  (defun my-mapa-b (fn a b &optional (step 1))
2    (map-> fn
3           a
4           #'(lambda(x) (> x b))
5           #'(lambda(X) (+ x step))))
```

Bibliografía

P. Graham.
ANSI Common Lisp.
Prentice Hall Series in Artificial Intelligence. Prentice Hall International, 1996.